

Denoising Autoencoder

An unsupervised network that learns to compress and reconstruct

Encoder/decoder architecture, the bottleneck, and what makes this 'denoising' specifically

01 What an Autoencoder Is (and isn't)

Every model up to this point was **supervised**: image in, digit label out, with a clear right answer to compare against. An autoencoder is different -- there's no label at all. Instead, the network's job is to take an image as input and reproduce that **same image** as output. This is called **self-supervised learning** -- the input doubles as its own target.

DEEP DIVE -- Why would reproducing the input be useful?

If the network could just copy pixels straight through, it would be trivial and useless. The trick is **forcing information through a narrow bottleneck** first. The network is built so the middle layer has far fewer neurons than the input (32 vs 784 here) -- so it's physically impossible to just pass data through unchanged. The network is forced to learn a **compressed representation** that still contains enough information to reconstruct the original reasonably well.

This compressed representation is useful on its own -- it's a learned, dense summary of the image's essential content, useful for compression, visualization, anomaly detection, or as input features to another model.

02 The 'Denoising' Twist

Adding artificial noise to the input

```
noise_factor = 0.5
X_train_noisy = X_train + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=X_train.shape)
X_train_noisy = np.clip(X_train_noisy, 0., 1.)
```

Argument	Value	Meaning
noise_factor	0.5	Scales how much noise gets added -- higher = noisier, harder reconstruction task
np.random.normal(loc=0.0, scale=1.0)	Gaussian noise, mean 0, std 1	Generates random values centered at 0 -- some pixels get brighter, some darker, randomly
np.clip(x, 0., 1.)	Clamps to [0, 1]	After adding noise, some pixel values fall below 0 or above 1 (invalid). Clipping forces them back into the valid image range

DEEP DIVE -- Why train on noisy input but clean target -- this is the entire point

Look closely at the fit() call:

```
autoencoder.fit(X_train_noisy, X_train, epochs=30, batch_size=256)
```

The **input** is the noisy image (X_train_noisy), but the **target** the loss compares against is the original clean image (X_train). The network never sees a clean version of its input during training -- it only ever sees noisy images, and is scored on how close it gets back to the clean original.

This forces the network to learn what digits actually **look like** structurally, rather than just memorising pixel-for-pixel copying. It has to learn to ignore the random noise and reconstruct the underlying signal -- exactly the behavior you'd want from a real image-cleanup tool.

A plain (non-denoising) autoencoder would train on $X_{\text{train}} \rightarrow X_{\text{train}}$ (clean to clean), which can let the network cheat by learning a near-identity mapping. Denoising forces genuine feature learning.

03 Architecture -- Encoder and Decoder

```
input_img = Input(shape=(784,))
encoder = Dense(units=32, activation='relu')(input_img)
decoder = Dense(units=784, activation='sigmoid')(encoder)

autoencoder = Model(input_img, decoder)
```

Layer	Units	Activation	Role
encoder	32	ReLU	Compresses 784 pixels down to a 32-number summary -- the 'bottleneck'
decoder	784	Sigmoid	Expands the 32-number summary back up to 784 pixel values

DEEP DIVE -- Why sigmoid on the decoder (and ReLU on the encoder)

Sigmoid on the output squashes every output value into (0, 1) -- which exactly matches the normalized pixel range the images were preprocessed into. This makes sigmoid a natural fit for image reconstruction tasks where pixels are scaled to [0,1], much like it's a natural fit for binary probabilities in classification.

ReLU on the encoder serves the same role it always does -- nonlinearity that's cheap to compute and trains well. There's no requirement it be ReLU specifically; the key constraint is the bottleneck size (32), not the activation choice.

The compression ratio

784 pixels compressed into 32 numbers is roughly a **24.5x compression ratio**. This is aggressive -- expect reconstructions to be blurry/imperfect, losing fine detail while preserving the general shape and identity of each digit. This is the visible tradeoff of forcing information through such a narrow bottleneck.

04 Loss Function and Compilation

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Argument	Value	Meaning
optimizer	'adam'	Adaptive optimizer that automatically tunes the effective learning rate per parameter -- a more modern, often faster-converging alternative to plain SGD
loss	binary_crossentropy	Treats each of the 784 output pixels as an independent probability (0-1) and penalises how far the reconstructed pixel intensity is from the true pixel intensity

Why binary cross-entropy for pixel values, not class labels

Binary cross-entropy is normally associated with yes/no classification, but here it's repurposed: each pixel's normalized intensity (0-1) is treated like a 'probability of being lit up', and the loss measures how well the reconstructed intensity matches the true intensity, pixel by pixel, averaged across all 784 pixels. Mean Squared Error (MSE) is an equally common and often-equivalent choice for this kind of task -- both penalise the gap between predicted and true pixel intensities, just with slightly different curve shapes.

```
autoencoder.fit(X_train_noisy, X_train, epochs=30, batch_size=256)
```

Notice there's no `validation_split` here, unlike the classifier notebooks -- this version trains on the full training set without holding out a validation slice to monitor during training.

05 Extracting the Encoder Alone

```
encoder_model = Model(input_img, encoder)
encoder_model.summary()
```

DEEP DIVE -- The trick: reusing trained layers in a new Model

This is a subtle but important pattern. **encoder_model** is a brand new Model object, but it's built from the same Input and the same encoder Dense layer that are already part of the trained autoencoder. Since layers in Keras carry their learned weights with them wherever they're referenced, this new model automatically has all the trained weights -- there's no retraining involved.

This lets you reuse just the compression half of the network on its own, to get the 32-number 'fingerprint' of any image, without needing the decoder at all. This is the foundation of how autoencoders get used for things like similarity search or dimensionality reduction in practice.

06 Predicting and Visualizing the Three Stages

```
pred = autoencoder.predict(X_test_noisy) # full pipeline: noisy -> clean reconstruction
encoded_images = encoder_model.predict(X_test) # encoder only: clean -> 32-number summary
```

Note the asymmetry: **pred** is generated from noisy test images through the FULL autoencoder (testing the denoising ability). **encoded_images** is generated from clean test images through ONLY the encoder (just inspecting what the compressed representation looks like).

The three-row visualization

```
plt.subplot(3, 20, i+1) # Row 1: noisy input
plt.imshow(X_test_noisy[i].reshape(28,28))

plt.subplot(3, 20, i+1+20) # Row 2: the 32-number encoding, reshaped for display
plt.imshow(encoded_images[i].reshape(8,4))

plt.subplot(3, 20, 2*20+i+1) # Row 3: reconstructed output
plt.imshow(pred[i].reshape(28,28))
```

Why `reshape(8,4)` for the encoded image

The encoder outputs 32 numbers per image -- not naturally a 2D image shape. **reshape(8,4)** is purely a visualization trick: $8 \times 4 = 32$, so it arranges those 32 numbers into an 8x4 grid just so they can be displayed as a small grayscale image. This has no real spatial meaning -- it's just a convenient way to glance at the 32 learned values rather than printing a list of numbers.

07 Summary -- The Full Pipeline

Step	Input	Output	Purpose
Add noise	Clean image	Noisy image	Create the denoising challenge
Encoder	Noisy image (784)	Compressed code (32)	Force the network to learn essential structure
Decoder	Compressed code (32)	Reconstructed image (784)	Rebuild the clean image from the compressed code
Loss comparison	Reconstructed image	Original CLEAN image (not noisy!)	Train the network to remove noise, not memorise it

What to look for in your three-row visualization

Row 1 (noisy) should look visibly grainy/speckled. Row 3 (reconstructed) should look noticeably cleaner than row 1, even if slightly blurry compared to a truly clean digit -- that blur is the visible cost of the 32-number bottleneck. Row 2 (the encoded representation) won't look like anything recognisable as a digit -- it's an abstract internal code, not meant to be human-interpretable.